Amiga Mail

Volume II

November/December 1990

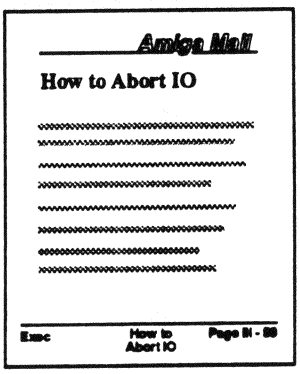
The Commodore Amiga Technical Newsletter

Copyright 1990 Commodore-Amiga, Inc. All rights reserved. Some of the materials used herein have been reproduced with the permission of the authors indicated. Use or reproduction of such materials shall be in accordance with the authors' instructions.

Amiga Mail
Volume II

Amiga Mail is published six times annually by the Commodore Applications and Technical Support Group (CATS). Its purpose is to support third-party developers of Amiga software and hardware worldwide by providing technical information and programming guidelines. Each issue features a collection of news and technical articles of special interest to Amiga product developers. The technical articles in each issue can be saved for future use in the Amiga Mail Tech Notes binder, a reference notebook containing all the articles from past issues of Amiga Mail.

For filing technical articles in the Amiga Mail Tech Notes binder, just use the chapter and page numbers listed in the footer of each article page. The Tech Notes chapter is shown as a Roman numeral followed by the page number within that chapter.



For instance, the article at right would be filed in chapter III of the Amiga Tech Notes binder, "Exec". The article starts on page 99 of that chapter in the binder.

For more information on the Amiga Mail Tech Notes binder and the Commodore Developer Support Program, contact your local support manager or if you are in the U.S., contact CATS at (215) 431-9180.

Warning

The information contained herein is subject to change without notice. Commodore specifically does not make any endorsement or representation with respect to the use, results, or performance of the information (including without limitation its capabilities, appropriateness, reliability, currentness or availability).

Disclaimer

This information is provided "as is" without warranty of any kind, either express or implied. The entire risk as to the use of this information is assumed by the user. In no event will Commodore or its affiliated companies be liable for any damages, direct, indirect, incidental, special or consequential, resulting from any defect in the information, even if advised of the possibility of such damages.

Amiga Mail News

DevCon Logo Contest

The 1991 North American Developers Conference will be held from September 4-7 in Denver, Colorado. Further information on the conference will be available at a later date.

CATS is sponsoring a DevCon Logo Contest open to developers registered in either the certified or commercial program. The winner will receive free admission to the conference as well as probable adulation and notoriety throughout the Amiga community. The winning logo will be used initially on developer conference materials, such as pre-DevCon mailings, registration materials, DevCon notes, T-shirts, etc.

Rules:

Each entry must be wholly the product of the registered developer in whose name the entry has been submitted or in the public domain. Copyrighted images are not eligible, including without limitation record album covers, movie figures, famous paintings, cartoon characters, etc. Outside assistance is expressly prohibited and will provide cause for disqualification.

Each entry must be created on the Amiga. A maximum of four colors (three colors and black) can be used.

The following elements should be incorporated in the design:

- North American Amiga DevCon
- Denver
- 1991

There is no limit to the number of entries each developer can submit.

Please include the following information with your entries: name, company name, developer number, address, phone. All submissions must be on disk in either ILBM or Professional Draw file format.

All entries must be postmarked by March 29, 1991. Please send all submissions to:

CATS - Logo Contest
1200 Wilson Drive
West Chester, PA 19380

General Conditions:

Entries will be judged (by employees of Commodore Applications and Technical Support) on design originality, creativity and suitability to theme.

Make sure your mailer will protect your disk from damage. Affix sufficient first class postage. Commodore assumes no responsibility for lost or damaged entries.

Entries and all proprietary rights to entries will become the property of Commodore Business Machines, Inc. automatically on receipt of entries. Commodore Business Machines reserves the right to adapt, use, publish or not use all entries received. No entries will be returned.

All rights to the Commodore and Amiga logos, and all other proprietary marks relating to Commodore

IN THIS ISSUE

Executing External Programs with Release 2.0
Carolyn Scheppner

AmigaDOS Packet Interface Specification
John Toebes

The DR2D FORM
Ross Cunniff and John Orr

products and services, remain the property of Commodore and its affiliates, and no entrant will acquire any license or other proprietary or contract right by entering or winning the contest. Non-winning entries will not be acknowledged. Winner will be notified by April 30, 1991. Employees of Commodore Business Machines, Inc., Commodore Electronics Ltd., Commodore International Ltd., their advertising and promotional agencies and their immediate families are not eligible to enter this contest.

NewTek Ships Toaster

NewTek, Inc., a Topeka, Kansas based manufacturer of Commodore Amiga software and peripherals for the video graphics market, announced it has begun shipping its ground breaking $1,595 (suggested retail) Video Toaster to computer and video dealers in the U.S.

A broadcast-quality video production board that fits into the video slot of Commodore Amiga 2000 series personal computers, the Video Toaster is revolutionary in its ability to combine core post-production tools in an inexpensive personal computing environment. Output from the Video Toaster is acceptable for television broadcast applications and rivals that of dedicated machines costing $10,000 - $50,000, according to NewTek officials.

"Video professionals who have seen the Toaster in action concur that it will set a new standard and direction for the video graphics industry", said Commodore Director of Business Markets David Archambault. "It's possibly the most exciting and strategically important third-party product ever developed for the Amiga."

"The Toaster puts broadcast-quality video output within reach of *any* serious user", Archambault added". "The Toaster is unique to the Amiga because it takes full advantage of the Amiga's custom coprocessors."

Key Video Toaster features include integrated production switching, digital effects, character generation and genlocking (the ability to overlay graphics onto a video signal). NewTek is also including paint and animation programs with the product.

Video Toaster applications include:

- Broadcast titling, effects and transitions.
- Video graphics/Post-production use
- Real-time graphics, titles and effects for multimedia presentation.

Midwest Communications, the U.S.'s largest professional video dealer, is among the many resellers which will distribute the product initially. Midwest expects to sell the Toaster and Amiga systems as turnkey solutions for specific applications in the independent, corporate, cable and broadcast video production markets.

Amiga World Announces New Technical Magazine

IDG Communications has announced the *AmigaWorld Technical Journal*, a bi-monthly publication that will focus on the needs of Amiga programmers. Publication will begin in March.

The magazine's articles are geared towards the serious programmer. Some of the programming projects will cover subjects such as designing libraries, maintaining compatibility with release 2.0, and controlling the new (ECS) video modes. In addition to the C and 680x0 assembler programming languages, articles covering programming in ARexx, Modula-2, and BASIC are planned.

The magazine comes with a companion disk. This disk will contain Public Domain utilities and applications as well as all the example program listings (with executables).

Other features include a news column, debates on important programming standards and issues, reviews of the latest commercial tools, and even hardware projects. A one-year introductory subscription costs $59.95. Single issues will be available for $14.95.

Executing External Programs with Release 2.0

by Carolyn Scheppner

The 2.0 Amiga operating system provides more flexible methods of launching programs from an application. The 2.0 *dos.library* contains a new function, System(), which is more flexible and powerful than the 1.3 Execute() function. Improvements to the con-handler, CON:, allow programs to create "auto con" windows, which are console windows that will only appear if input is requested from or output is sent to that console. The 2.0 con-handler also provides several other enhancements, including Close gadgets for CON: windows and a feature which allows a program to use an existing window as its CON: window.

The System() Function

The System() function is intended to replace the Execute() function of release 1.3. System() spawns a shell process to execute the command line which is passed to System() as an argument. The shell parses the command-line normally, just like command-lines typed directly into the shell's console. If it can, System() will pass the current path and directory to the programs it launches.

The System() function can execute external commands either synchronously or asynchronously. When System() is used synchronously, control returns to the calling program after the external program has completed. In this case, System() returns the external program's return code, or a -1 if the command could not be found or run. On the other hand, when System() initiates a program asynchronously, the program is no longer a concern for the caller. The operating system will take care of the cleanup. This is extremely useful for an application that must start up multiple programs on user demand, such as a hot key commodity. By default, System() starts programs synchronously. To launch a program asynchronously, use the `SYS_Asynch` tag with the data field set to `TRUE`.

With the System() call, it is easy to provide programs with specific input and output handles. The tags `SYS_Input` and `SYS_Output` (defined in *dos/dostags.h*) are used to supply the input and output file handles. A program can pass its own input and output handles to a *synchronously* launched program by passing the results of Input() and Output(), respectively. Note that with a synchronous System() call, the OS will not close these handles when the spawned process exits. In the case of an asynchronously launched program, the launching program normally must provide *new* IO handles since the system automatically closes these handles when the asynchronous process ends. Because AmigaDOS wants separate handles for input and output, System() will automatically create an output handle if it's passed a handle for `SYS_Input` and `NULL` for `SYS_Output`. This allows a program to open a CON: window for input and use it for both input and output, as System() will test to see if input is interactive and, if so, will attempt to open "*" for output to that console. If the input file handle is not interactive, System() opens "*" on the current console task.

Programs launched using System() are not restricted to the built-in, or Boot, shell. In the near future, System() will be able to take advantage of specially designed shells (the design requirements of these special shells have not yet been documented). Two tags, `SYS_UserShell` and `SYS_CustomShell`, specify which shell System() should use to execute the command-line. By using the `SYS_UserShell` tag with a tag value of `TRUE`, an application tells System() to send a command to the user's preferred shell rather that the boot shell (Note that

the default user shell is the boot shell). If an application opens a shell for the user or executes the user's command-lines, it should tell System() to use the user shell. If an application requires consistent shell behavior, it must not use the user shell because the user can change the user shell. Another shell tag, `SYS_CustomShell`, allows an application to choose other shells besides the boot and user shells. This tag's data field should contain the custom shell's name as it appears in the system resident list.

Although it does offer many features, System() does have some limitations. First, because command paths currently only exist in the CLI structure, Workbench (non-CLI) processes have no paths. Consequently, when a Workbench process calls a program using System(), the shell has no path to search for that program (aside from the system default search path of *C:* and the current directory). A second limitation of the System() function involves CTRL-C/D/E/F handling. When a task opens a CON: window to provide a handle for a System()-launched command, that task is the owner of the handle. This has two effects. A System()-launched command running in that CON: window will only be able to receive CTRL-C/D/E/F signals while it is doing input or output to the window (i.e. when it has a pending read or write). If the task that owns the handle is still around, it receives CTRL-C/D/E/F signals whenever those keys are pressed in the CON: window.

The 2.0 Con-handler

The 2.0 con-handler (CON:) has many new enhancements that allow programs to further customize their console windows. An application requests these new features by appending keywords to the end of the CON: specification string for Open(). These keywords may appear in any order *after* the title string in the CON: specification.

These new keywords are:

`AUTO`		Don't open CON: window until/unless input or output occurs
`CLOSE`		Put a close gadget on the CON: window
`WAIT`		Hold off Close until user clicks Close or types CTRL-\
`WINDOW`	`0xaddr`	Use specified window (may be on a custom screen)
`SCREEN`	`name`	Open on specified public screen

The additional CON: keywords BACKDROP, NODRAG, NOBORDER, NOSIZE, SIMPLE, and SMART, allow control of other attributes of a CON: window.

An AUTO/CLOSE/WAIT CON: window is perfect for 2.0-specific C startup code and for asynchronous System() startup of arbitrary commands. Because of the "auto con" feature (AUTO), the system will never open the CON: window of a program that doesn't do any stdio (example: Calculator). If a command does stdio input or output, the window will not open until stdio occurs. If the window opens, the wait feature (WAIT) causes the window to stay open until the user types CTRL-\ (end of file) or clicks the Close gadget. The CLOSE keyword tells the con-handler to put a close gadget on the CON: window.

Example: `"CON:/0/0/640/200/My Title/AUTO/CLOSE/WAIT"`

The SCREEN keyword along with a public screen name allows an application to open a CON: window on that public screen. Alternately, an application can use the WINDOW keyword to attach a console to an already open Intuition window. The hex address of the Intuition window must follow the WINDOW keyword. This makes it possible for System()-launched programs to do their stdio in a window on a custom screen.

The example code, *SystemTest.c*, provides two simple subroutines for executing external commands, and demonstrates the following 2.0 features:

- Synchronous System() command execution using the calling program's Input()/Output().
- Synchronous System() command execution in a custom screen window.
- Asynchronous System() command startup with an AUTO/CLOSE/WAIT window.
- OpenScreenTags and OpenWindowTags for a 2.0 New Look window.

```
;/* SystemTest.c - Execute me to compile me with Lattice 5.04
;   Demonstration of System(), AUTO CON, and custom screen CON
LC -bl -cfistq -v -y -j73 SystemTest.c
Blink FROM LIB:c.o,SystemTest.o TO SystemTest LIBRARY LIB:LC.lib,LIB:Amiga.lib
quit
*/

#include <exec/types.h>
#include <exec/libraries.h>
#include <dos/dos.h>
#include <dos/dostags.h>
#include <intuition/intuition.h>
#include <graphics/displayinfo.h>

#ifdef LATTICE
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/intuition_protos.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }  /* really */
#endif

/* our function error codes */
#define SYSTEMFAIL      (-1L)
#define WINDOWFAIL      (-2L)

/* function prototypes */
LONG beginCommand(UBYTE *command);
LONG doCommand(UBYTE *command, BPTR other);
VOID checkResult(UBYTE *command, LONG result);


/* Formatted version string for the 2.0 VERSION command */
UBYTE *vers = "\0$VER: SystemTest 36.10";

struct Library *IntuitionBase;


struct Screen *OpenScreenTags(struct NewScreen *,ULONG tags, ...);
struct Window *OpenWindowTags(struct NewWindow *,ULONG tags, ...);


void main(int argc, char **argv)
    {
    extern struct Library *DOSBase;
    struct Screen *scr = NULL;
    struct Window *win = NULL;
    ULONG penspecterm = ~0;
    LONG result;
    BPTR file;
    UBYTE *command;
    UBYTE buf[128];

    if(DOSBase->lib_Version < 36)
        {
        printf("This example requires dos.library V36 or higher\n");
        exit(RETURN_FAIL);
        }
    if(!(IntuitionBase = OpenLibrary("intuition.library",36)))
        {
        printf("This example requires intuition.library V36 or higher\n");
        exit(RETURN_FAIL);
        }

    /* SYNCHRONOUS SYSTEM() WITH OUR INPUT/OUTPUT
     */
    printf("\n*** SystemTest: Synchronous System call 'dir libs:':\n");
    command = "dir libs:";
    result = doCommand(command,NULL);
    checkResult(command,result);

    printf("\n*** SystemTest: Synchronous System call of nonexistant command:\n");
    command = "badcommand";
    result = doCommand(command,NULL);
    checkResult(command,result);

    printf("\n*** SystemTest: Synchronous System call 'ask \"...Answer y now\"':\n");
    command =
        "ask \"Ready for CON: on a Custom Screen? Answer y now (should return 5):\"";
    result = doCommand(command,NULL);
    checkResult(command,result);

   /* SYNCHRONOUS SYSTEM() WITH CON: IN A CUSTOM SCREEN AND WINDOW
    */
    if(scr = OpenScreenTags(NULL,
                SA_Width, 640,
                SA_Height, 200,
                SA_Depth, 3,
                SA_DisplayID, HIRES_KEY,
                SA_Pens, &penspecterm,  /* Give us New Look */
                TAG_DONE))
        {
        if(win = OpenWindowTags(NULL,
                WA_CustomScreen, scr,
                WA_Flags, WINDOWDRAG|WINDOWCLOSE|ACTIVATE,
                WA_IDCMP, CLOSEWINDOW,
                WA_Top, 20,
                WA_Height, scr->Height - 20,
                WA_Title, "Custom Window",
                WA_ScreenTitle, "Custom Screen",
                TAG_DONE))
            {
            sprintf(buf,"CON:////WINDOW0x%lx", win); /* adds window pointer */
            if(file = Open(buf, MODE_OLDFILE))
                {
                command = "echo \"CLI commands on a custom screen!\"";
                result = doCommand(command,file);
                command = "dir libs:";
                result = doCommand(command,file);
                command = "echo \"( Click CLOSE gadget, or type CTRL-C )\"";
                result = doCommand(command,file);
                Wait(1 << win->UserPort->mp_SigBit | SIGBREAKF_CTRL_C);
                Close(file);    /* Closes the window too */
                }
            else CloseWindow(win);
            }
        CloseScreen(scr);
        }

    printf("\n*** SystemTest: Synchronous System call 'ask \"...Answer y now\"':\n");
    command = "ask \"Ready for Asynchronous demo? Answer y now (should return 5):\"";
    result = doCommand(command,NULL);
    checkResult(command,result);


    /* ASYNCHRONOUS SYSTEM() WITH ON-DEMAND AUTO/WAIT CON:
     */
    printf("\n*** SystemTest: Asynchronous startup of 'Sys:Utilities/Clock':\n");
    command = "SYS:Utilities/Clock";
    result = beginCommand(command);
    checkResult(command,result);

    printf("\n*** SystemTest: Asynchronous startup of 'avail':\n");
    command = "avail";
    result = beginCommand(command);
    checkResult(command,result);

    printf("\nSystemTest exiting. Close Clock and Autocon window when you wish.\n");

    CloseLibrary(IntuitionBase);
    exit(RETURN_OK);
    }
```

```
/*
 * Synchronous external command (wait for return)
 * Uses your Input/Output unless you supply other handle
 * Result will be return code of the command, unless the System() call
 * itself fails, in which case the result will be -1
 */
LONG doCommand(UBYTE *command, BPTR other)
    {
    struct TagItem stags[4];

    stags[0].ti_Tag = SYS_Input;
    stags[0].ti_Data = other ? other : Input();
    stags[1].ti_Tag = SYS_Output;
    stags[1].ti_Data = other ? NULL: Output();
    stags[3].ti_Tag = TAG_DONE;
    return(System(command, stags));
    }



/*
 * Asynchronous external command started with its own autocon Input/Output
 * This routine shows use of the SYS_UserShell tag as well.
 * Result will only reflect whether System() call itself succeeded.
 * If System() call fails, result will be -1L
 * We are using -2L as result if our Open of CON: fails
 */
UBYTE *autocon="CON:0/40/640/150/Auto CON Window Opens if Needed/auto/close/wait";
LONG beginCommand(UBYTE *command)
    {
    struct TagItem stags[5];
    BPTR file;

    if(file = Open(autocon, MODE_OLDFILE))
        {
        stags[0].ti_Tag = SYS_Input;
        stags[0].ti_Data = file;
        stags[1].ti_Tag = SYS_Output;
        stags[1].ti_Data = NULL;
        stags[2].ti_Tag = SYS_Asynch;
        stags[2].ti_Data = TRUE;
        stags[3].ti_Tag = SYS_UserShell;
        stags[3].ti_Data = TRUE;
        stags[4].ti_Tag = TAG_DONE;
        return(System(command, stags));
        }
    else return(WINDOWFAIL);
    }



/*
 * Demo routine outputs result of System
 */
VOID checkResult(UBYTE *command, LONG result)
    {
    if(result == SYSTEMFAIL)
        printf("*** SystemTest: could not start process for command\n");
    else if(result == WINDOWFAIL)
        printf("*** SystemTest: can't open con: for command\n");
    else
        printf("*** SystemTest: command (if synchronous) returned %ld\n",result);
    }
```

```
/*
 * Stack based stubs for opening screen, window
 */
struct Screen *OpenScreenTags(ns, tags)
struct NewScreen *ns;
ULONG tags;
    {
    return (OpenScreenTagList(ns, (struct TagItem *)&tags));
    }

struct Window *OpenWindowTags(nw, tags)
struct NewWindow *nw;
ULONG tags;
    {
    return (OpenWindowTagList(nw, (struct TagItem *)&tags));
    }
```

AmigaDOS Packet Interface Specification

by John Toebes

AmigaDOS communicates with file systems and other DOS handlers by sending and receiving *packets*. Opening and closing file handles (including console file handles), creating directories, and renaming disks all require DOS to tell a handler to perform these actions through sending a packet. The particular action a handler performs depends on the type of packet it receives.

This article documents the standard AmigaDOS packet types. For information on how to use packets to communicate with handlers see the *AmigaDOS Manual*.

Packets sent to a file system or handler can be divided into several basic categories:

- **Basic Input/Output**

These actions deal with tranferring data to and from objects controlled by the handler.

- **File/Directory Manipulation/Information**

These actions are used to gain access to and manipulate the high level structures of the file system.

- **Volume Manipulation/Information**

These actions allow access to the specific volume controlled by the file system.

- **Handler Maintenance and Control**

These allow control over the handler/file system itself, independent of the actual volume or structure underneath.

- **Handler Internal**

These actions are never sent to the handler directly. Instead they are generally responses to IO requests made by the handler. The handler makes these responses look like packets in order to simplify processing.

- **Obsolete Packets**

These packets are no longer valid for use by handlers and file systems.

- **Console Only Packets**

These packets are specific to console handlers. File Systems can ignore these packets.

Much of this information can be extracted from Developer Conference notes , *The AmigaDOS Manual*, and various Fred Fish disks. However, because there is no single complete reference to these packet types, a consolidated view of all the packets is presented here. Several structures are referenced here which can be found by looking at the include files <*dos/dos.h*> and <*dos/dosextens.h*>. (If you are using the 1.3 version of the include files, these are in the *libraries* directory instead of the *dos* directory). Before attempting to work with a file handler you should first become familiar with these files.

Each packet type documented in this article is listed with its action name, its corresponding number, any AmigaDOS routines which uses this packet, and the list of parameters that the packets uses. The C variable types for the packet parameters are one of the following types:

BPTR This is BCPL pointer (the address of the given object shifted right by 2). Note: this means that the object must be aligned on a longword boundary.

LOCK This is a BPTR to a FileLock structure returned by a previous `ACTION_LOCATE_OBJECT`. A lock of 0 is legal, *indicating the root of the volume for the handler.*

BSTR This is a BPTR to a string where the first byte indicates the number of characters in the string. This length byte is unsigned but because it is stored in a byte, the strings are limited to 255 characters in length.

BOOL A 32-bit boolean value either containing `DOSTRUE` (-1) or `DOSFALSE` (0). Note: equality comparisons with `DOSTRUE` should be avoided.

CODE A 32 bit error code as defined in the *dos/dos.h* include file. Handlers should not return error codes besides those defined in *dos/dos.h*.

ARG1 The FileHandle->fh_Arg1 field.

LONG A 32 bit integer value.

Basic Input/Output

The Basic Input/Output actions are supported by both handlers and file systems. In this way, the application can get a stream level access to both devices and files. One difference that arises between the two is that a handler will not necessarily support an ACTION_SEEK while it is generally expected for a file system to do so.

These actions work based on a FileHandle which is filled in by one of the three forms of opens:

```
ACTION_FINDINPUT          1005    Open(..., MODE_OLDFILE)
ACTION_FINDOUTPUT         1006    Open(..., MODE_NEWFILE)
ACTION_FINDUPDATE         1004    Open(..., MODE_READWRITE)
ARG1:  BPTR FileHandle to fill in
ARG2:  LOCK Lock on directory that ARG3 is relative to
ARG3:  BSTR Name of file to be opened (relative to ARG1)

RES1:  BOOL Success/Failure (DOSTRUE/DOSFALSE)
RES2:  CODE Failure code if RES1 is DOSFALSE
```

All three actions use the lock (ARG2) as a base directory location from which to open the file. If this lock is NULL, then the file name (ARG3) is relative to the root of the current volume. Because of this, file names are not limited to a single file name but instead can include a volume name (followed by a colon) and multiple slashes allowing the file system to fully resolve the name. This eliminates the need for AmigaDOS or the application to parse names before sending them to the file system. Note that the lock in ARG2 must be associated with the file system in question. It is illegal to use a lock from another file system.

The calling program owns the file handle (ARG1). The program must initialize the file handle before trying to open anything (in the case of a call to Open(), AmigaDOS allocates the file handle automatically and then frees it in Close()). All fields must be zero except the fh_Pos and fh_End fields which should be set to -1. Upon a successful open, the *handler* is responsible for filling in the fh_Type field with a pointer to the MsgPort of the handler process. Lastly, the handler must initialize fh_Arg1 with something that allows the handler to uniquely locate the object being opened (normally a file). This value is implementation specific. This field is passed to the READ/WRITE/SEEK/ END/TRUNCATE operations and not the file handle itself.

FINDINPUT and FINDUPDATE are similar in that they only succeed if the file already exists. FINDINPUT will open with a shared lock while FINDUPDATE will open it with a shared lock but if the file doesn't exist, FINDUPDATE will create the file. FINDOUTPUT will always open the file (deleting any existing one) with an exclusive lock.

```
ACTION_READ               'R'     Read(...)
ARG1:  ARG1 fh_Arg1 field of the opened FileHandle
ARG2:  APTR Buffer to put data into
ARG3:  LONG Number of bytes to read

RES1:  LONG Number of bytes read.
        0 indicates EOF.
       -1 indicates ERROR
RES2:  CODE Failure code if RES1 is -1
```

This action extracts data from the file (or input channel) at the current position. If fewer bytes

remain in the file than requested, only those bytes remaining will be returned with the number of bytes stored in RES1. The handler indicates an error is indicated by placing a -1 in RES1 and the error code in RES2. If the read fails, the current file position remains unchanged. Note that a handler may return a smaller number of bytes than requested, even if not at the end of a file. This happens with interactive type file handles which may return one line at a time as the user hits return, for example the console handler, CON:.

```
ACTION_WRITE                'W'     Write(...)
ARG1:  ARG1 fh_Arg1 field of the opened file handle
ARG2:  APTR Buffer to write to the file handle
ARG3:  LONG Number of bytes to write

RES1:  LONG Number of bytes written.
RES2:  CODE Failure code if RES1 not the same as ARG3
```

This action copies data into the file (or output channel) at the current position. The file is automatically extended if the write passes the end of the file. The handler indicates failure by returning a byte count in RES1 that differs from the number of bytes requested in ARG3. In the case of a failure, the handler does not update the current file position (although the file may have been extended and some data overwritten) so that an application can safely retry the operation.

```
ACTION_SEEK                 1008    Seek(...)
ARG1:  ARG1     fh_Arg1 field of the opened FileHandle
ARG2:  LONG     New Position
ARG3:  LONG     Mode:   OFFSET_BEGINNING,OFFSET_END, or  OFFSET_CURRENT

RES1:  LONG     Old Position.   -1 indicates an error
RES2:  CODE     Failure code if RES1 = -1
```

This packet sets the current file position. The new position (ARG2) is relative to either the beginning of the file (`OFFSET_BEGINNING`), the end of the file (`OFFSET_END`), or the current file position (`OFFSET_CURRENT`), depending on the mode set in ARG3. Note that ARG2 can be negative. The handler returns the previous file position in RES1. Any attempt to seek past the end of the file will result in an error and will leave the current file position in an unknown location.

```
ACTION_END                  1007    Close(...)
ARG1:   ARG1    fh_Arg1 field of the opened FileHandle

RES1:   LONG    DOSTRUE
```

This packet closes an open file handle. This function generally returns a `DOSTRUE` as there is little the application can do to recover from a file closing failure. If an error is returned under 2.0, DOS will not deallocate the file handle. Under 1.3, it does not check the result.

2.0 only

```
ACTION_LOCK_RECORD       2008    LockRecord(fh,pos,len,mod,tim)
ARG1:   BPTR    FileHandle to lock record in
ARG2:   LONG    Start position (in bytes) of record in the file
ARG3:   LONG    Length (in bytes) of record to be locked
ARG4:   LONG    Mode
                 0 = Exclusive
                 1 = Immediate Exclusive (timeout is ignored)
                 2 = Shared
                 3 = Immediate Shared (timeout is ignored)
ARG5:   LONG    Timeout period in AmigaDOS ticks (0 is legal)

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 is DOSFALSE
```

This function locks an area of a file in either a sharable (indicating read-only) or exclusive (indicating read/write) mode. Several sharable record locks from different file handles can exist simultaneously on a particular file area but only one file handle can have exclusive record locks on a particular area at a time. The "exclusivity" of an exclusive file lock only applies to record locks from other file handles, not to record locks within the file handle. One file handle can have any number of overlapping exclusive record locks. In the event of overlapping lock ranges, the entire range must be lockable before the request can succeed. The timeout period (ARG5) is the number of AmigaDOS ticks (1/50 second) to wait for success before failing the operation.

2.0 only

```
ACTION_FREE_RECORD       2009    FreeRecord(file,pos,len)
ARG1:   BPTR    FileHandle to unlock record in
ARG2:   LONG    Start position (in bytes) of record in the file
ARG3:   LONG    Length of record (in bytes) to be unlocked

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 is DOSFALSE
```

This function unlocks any previous record lock. If the given range does not represent one that is currently locked in the file, ACTION_FREE_RECORD returns an error. In the event of multiple locks on a given area, only one lock is freed.

2.0 only

```
ACTION_SET_FILE_SIZE     1022    SetFileSize(file,off,mode)
ARG1:   BPTR    FileHandle of opened file to modify
ARG2:   LONG    New end of file location based on mode
ARG3:   LONG    Mode.  One of OFFSET_CURRENT, OFFSET_BEGIN, or OFFSET_END

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 is DOSFALSE
```

This function is used to change the physical size of an opened file. ARG2, the new end-of-file position, is relative to either the current file position (OFFSET_CURRENT), the beginning of the file (OFFSET_BEGIN), or the end of the file (OFFSET_END), depending on the mode set in ARG3. The current file position will not change unless the current file position is past the new end-of-file position. In this case, the new file position will move to the new end of the file. If there are other open file handles on this file, ACTION_SET_FILE_SIZE sets the end-of-file for these alternate file handles to either their respective current file position or to the new end-of-file position of the file handle in ARG1, whichever makes the file appear longer.

Directory/File Manipulation/Information

The directory/file actions permits an application to make queries about and modifications to handler objects. These packets perform functions such as creating subdirectories, resolving links, and filling in FileInfoBlock structures for specific files.

```
ACTION_LOCATE_OBJECT      8       Lock(...)
ARG1:   LOCK    Lock on directory to which ARG2 is relative
ARG2:   BSTR    Name (possibly with a path) of object to lock
ARG3:   LONG    Mode:   ACCESS_READ/SHARED_LOCK, ACCESS_WRITE/EXCLUSIVE_LOCK

RES1:   LOCK    Lock on requested object or 0 to indicate failure
RES2:   CODE    Failure code if RES1 = 0
```

The AmigaDOS function Lock() uses this action to create its locks. Given a name for the object, which may include a path, (ARG2) and a lock on a directory from which to look for the name (and path), ACTION_LOCATE_OBJECT will locate the object within the file system and create a FileLock structure associated with the object. If the directory lock in ARG1 is NULL, the name is relative to the root of the file handler's volume (a.k.a. ":"). The memory for the FileLock structure returned in RES1 is maintained by the handler and freed by an ACTION_FREE_LOCK. Although it's not a requirement, if an handler expects to support the pre-1.3 *Format* command, it must accept any illegal mode as ACCESS_READ.

A handler can create an exclusive lock only if there are no other outstanding locks on the given object. Once created, an exclusive lock prevents any other locks from being created for that object. In general, a handler uses the FileLock->fl_Key field to uniquely identify an object. Note that some applications rely on this (although a handler is not required to implement this packet).

The fl_Volume field of the returned FileLock structure should point to the DOS device list's volume entry for the volume on which the lock exists. In addition, there are several diagnostic programs that expect all locks for a volume to be chained together off the dl_LockList field in the volume entry. *Note that relying on this chaining is **not** safe, and can cause serious problems including a system crash. No application should use it.*

```
ACTION_COPY_DIR           19      DupLock(...)
ARG1:   LOCK    Lock to duplicate

RES1:   LOCK    Duplicated Lock or 0 to indicate failure
RES2:   CODE    Failure code if RES1 = 0
```

This action's name is misleading as it does not manipulate directories. Instead, it creates a copy of a *shared* lock. The copy is subsequently freed with an ACTION_FREE_LOCK. Note that it is valid to pass a NULL lock. Currently, the DupLock() call always returns 0 if passed a 0, although a handler is not required to return a 0.

```
ACTION_FREE_LOCK          15      UnLock(...)
ARG1:   LOCK    Lock to free

RES1:   BOOL    TRUE
```

This action frees the lock passed to it. The AmigaDOS function Unlock() uses this packet. If passed a NULL lock, the handler should return success.

```
ACTION_EXAMINE_OBJECT    23       Examine(...)
ARG1:    LOCK    Lock of object to examine
ARG2:    BPTR    FileInfoBlock to fill in

RES1:    BOOL    Success/failure (DOSTRUE/DOSFALSE)
RES2:    CODE    Failure code if RES1 = DOSFALSE
```

This action fills in the FileInfoBlock with information about the locked object. The Examine() function uses this packet. This packet is actually used for two different types of operations. It is called to obtain information about a given object while in other cases, it is called to prepare for a sequence of EXAMINE_NEXT operations in order to traverse a directory.

This seemingly simple operation is not without its quirks. One in particular is the FileInfoBlock->fib_Comment field. This field used to be 116 bytes long, but was changed to 80 bytes in release 1.2. The extra 36 bytes lie in the fib_Reserved field. Another quirk of this packet is that both the fib_EntryType and the fib_DirEntryType fields must be set to the same value, as some programs look at one field while other programs look at the other.

File systems should use the same values for fib_DirEntryType as the ROM file system and ram-handler do. These are as follows:

```
ST_ROOT          1
ST_USERDIR       2
ST_SOFTLINK      3 NOTE: this Shows up as a directory unless checked for explicitly
ST_LINKDIR       4
ST_FILE         -3
ST_LINKFILE     -4
```

Also note that for directories, handlers ***must*** use numbers greater than 0, since some programs test to see if fib_DirEntryType is greater than zero, ignoring the case where fib_DirEntryType equals 0. Handlers should avoid using 0 because it is not interpreted consistently.

```
ACTION_EXAMINE_NEXT      24       ExNext(...)
ARG1:    LOCK    Lock on directory being examined
ARG2:    BPTR    BPTR FileInfoBlock

RES1:    BOOL    Success/failure (DOSTRUE/DOSFALSE)
RES2:    CODE    Failure code if RES1 = DOSFALSE
```

The ExNext() function uses this packet to obtain information on all the objects in a directory. ACTION_EXAMINE fills in a FileInfoBlock structure describing the first file or directory stored in the directory referred to in the lock in ARG1. ACTION_EXAMINE_NEXT is used to find out about the rest of the files and directories stored in the ARG1 directory. ARG2 contains a pointer to a valid FileInfoBlock field that was filled in by either an ACTION_EXAMINE or a previous ACTION_EXAMINE_NEXT call. It uses this structure to find the next entry in the directory. This packets writes over the old FileInfoBlock with information on the next file or directory in the ARG2 directory. ACTION_EXAMINE_NEXT returns a failure code of ERROR_NO_MORE_ENTRIES when there are no more files or directories left to be examined. Unfortunately, like ACTION_EXAMINE, this packet has its own peculiarities. Among the quirks that ACTION_EXAMINE_NEXT must account for are:

• The situation where an application calls ACTION_EXAMINE_NEXT one or more times and then stops invoking it before encountering the end of the directory.

• The situation where a FileInfoBlock passed to ACTION_EXAMINE_NEXT is not the same as the one passed to ACTION_EXAMINE or even the previous EXAMINE_NEXT operation. Instead, it is a copy of the FileInfoBlock with only the fib_DiskKey and the first 30 bytes of the fib_FileName fields copied over. *This is now considered to be illegal and will not work in the future. Any new code should not be written in this manner.*

• Because a handler can receive other packet types between ACTION_EXAMINE_NEXT operations, the ACTION_EXAMINE_NEXT function must handle any special cases that may result.

• The LOCK passed to ACTION_EXAMINE_NEXT is not always the same lock used in previous operations. It is however a lock on the same object.

Because of these problems, ACTION_EXAMINE_NEXT is probably the trickiest action to write in any handler. Failure to handle any of the above cases can be quite disastrous.

```
ACTION_CREATE_DIR          22       CreateDir(...)
ARG1:    LOCK     Lock to which ARG2 is relative
ARG2:    BSTR     Name of new directory  (relative to ARG1)

RES1:    BOOL     Success/failure (DOSTRUE/DOSFALSE)
RES2:    CODE     Failure code if RES1 = DOSFALSE

ACTION_DELETE_OBJECT       16       DeleteFile(...)
ARG1:    LOCK     Lock to which ARG2 is relative
ARG2:    BSTR     Name of object to delete (relative to ARG1)

RES1:    BOOL     Success/failure (DOSTRUE/DOSFALSE)
RES2:    CODE     Failure code if RES1 = DOSFALSE

ACTION_RENAME_OBJECT       17       Rename(...)
ARG1:    LOCK     Lock to which ARG2 is relative
ARG2:    BSTR     Name of object to rename (relative to ARG1)
ARG3:    LOCK     Lock associated with target directory
ARG4:    BSTR     Requested new name for the object

RES1:    BOOL     Success/failure (DOSTRUE/DOSFALSE)
RES2:    CODE     Failure code if RES1 = DOSFALSE
```

These three actions perform most of the work behind the AmigaDOS commands *MakeDir*, *Delete*, and *Rename* (for single files). These packets take as their parameters a lock describing where the file is and a name relative to that lock. It is the responsibility of the file system to ensure that the operation is not going to cause adverse effects. In particular, the RENAME_OBJECT action allows moving files across directory bounds and as such must ensure that it doesn't create hidden directory loops by renaming a directory into a child of itself.

For Directory objects, the DELETE_OBJECT action must ensure that the directory is empty before allowing the operation.

```
ACTION_PARENT              29       Parent(...)
ARG1:    LOCK     Lock on object to get the parent of

RES1:    LOCK     Parent Lock
RES2:    CODE     Failure code if RES1 = 0
```

This action receives a lock on an object and creates a shared lock on the object's parent. If the original object has no parent, then a lock of 0 is returned. Note that this operation is typically used in the process of constructing the absolute path name of a given object.

```
ACTION_SET_PROTECT        21      SetProtection(...)
ARG1:   Unused
ARG2:   LOCK    Lock to which ARG3 is relative
ARG3:   BSTR    Name of object (relative to ARG2)
ARG4:   LONG    Mask of new protection bits

RES1:   BOOL    Success/failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 = DOSFALSE
```

This action allows an application to modify the protection bits of an object. The 4 lowest order bits (RWED) are a bit peculiar. If their respective bit is set, that operation is not allowed (i.e. if a file's delete bit is set the file is *not* deleteable). By default, files are created with the RWED bits set and all others cleared. Additionally, any action which modifies a file is required to clear the A (archive) bit. See the *dos/dos.h* include file for the definitions of the bit fields.

```
ACTION_SET_COMMENT        28      SetComment(...)
ARG1:   Unused
ARG2:   LOCK    Lock to which ARG3 is relative
ARG3:   BSTR    Name of object (relative to ARG2)
ARG4:   BSTR    New Comment string

RES1:   BOOL    Success/failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 = DOSFALSE
```

This action allows an application to set the comment string of an object. If the object does not exist then DOSFALSE will be returned in RES1 with the failure code in RES2. The comment string is limited to 79 characters.

```
ACTION_SET_DATE           34      SetFileDate(...) in 2.0
ARG1:   Unused
ARG2:   LOCK    Lock to which ARG3 is relative
ARG3:   BSTR    Name of Object (relative to ARG2)
ARG4:   CPTR    DateStamp

RES1:   BOOL    Success/failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 = DOSFALSE
```

This action allows an application to set an object's creation date.

2.0 only

```
ACTION_FH_FROM_LOCK       1026    OpenFromLock(lock)
ARG1:   BPTR    Lock of file to be opened
ARG2:   LONG    Mode for file handle - either, MODE_OLDFILE, MODE_NEWFILE,
or MODE_READWRITE

RES1:   BPTR    Created filehandle or NULL
RES2:   CODE    Failure code if RES1 = NULL
```

This action open a file from a given lock. If this action is successful, the file system will essentially steal the lock so a program should not use it anymore. If ACTION_FH_FROM_LOCK fails, the lock is still usable by an application.

2.0 only

```
ACTION_SAME_LOCK          40      SameLock(lock1,lock2)
ARG1:   BPTR    Lock 1 to compare
ARG2:   BPTR    Lock 2 to compare

RES1:   LONG    Result of comparison, one of
  LOCK_SAME           (0) if locks are for the same object
  LOCK_SAME_HANDLER   (1) if locks are on different objects of same handler
  LOCK_DIFFERENT     (-1) otherwise
RES2:   CODE    Failure code if RES1 is LOCK_DIFFERENT
```

This action compares the targets of two locks. If they point to the same object, ACTION_SAME_LOCK should return LOCK_SAME.

2.0 only

```
ACTION_MAKE_LINK          1021    MakeLink(name,targ,mode)
ARG1:   BPTR    Lock on directory ARG2 is relative to
ARG2:   BSTR    Name of the link to be created (relative to ARG1)
ARG3:   BPTR    Lock on target object or name (for soft links).
ARG4:   LONG    Mode of link, either LINK_SOFT or LINK_HARD

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 is DOSFALSE
```

This packet causes the file system to create a link to an already existing file or directory. There are two kinds of links, hard links and soft links. The basic difference between them is that a file system resolves a hard link itself, while the file system passes a string back to DOS telling it where to find a soft linked file or directory. To the packet level programmer, there is essentially no difference between referencing a file by its original name or by its hard link name. In the case of a hard link, ARG3 is a lock on the file or directory that the link is "linked" to, while in a soft link, ARG3 is a BSTR naming the link.

In an over-simplified model of the ROM file system, when asked to locate a file, the system scans a disk looking for a file header with a specific (file) name. That file header points to the actual file data somewhere on the disk. With hard links, more than one file header can point to the same file data, so data can be referenced by more than one name. When the user tries to delete a hard link to a file, the system first checks to see if there are any other hard links to the file. If there are, only the hard link is deleted, the actual file data the hard link used to reference remains, so the existing hard links can still use it. In the case where the original link (not a hard or soft link) to a file is deleted, the file system will make one of its hard links the new "real" link to the file. Hard links can exist on directories as well. Because hard links "link" directly to the underlying media, hard links in one file system cannot reference objects in another file system.

Soft links are resolved through DOS calls. When the file system scans a disk for a file or directory name and finds that the name is a soft link, it returns an error code (ERROR_IS_SOFT_LINK). If this happens, the application must ask the file system to tell it what the link the link refers to by calling ACTION_READ_LINK. Soft Links are stored on the media, but instead of pointing directly to data on the disk, a soft link contains a path to its object. This path can be relative to the lock in ARG1, relative to the volume (where the string will be prepended by a colon ':'), or an absolute path. An absolute path contains the name of another volume, so a soft link can reference files and directories on other disks.

2.0 only

```
ACTION_READ_LINK          1024  ReadLink(port,lck,nam,buf,len)
ARG1:   BPTR    Lock on directory that ARG2 is relative to
ARG2:   CPTR    Path and name of link (relative to ARG1).  NOTE: This is a C
string not a BSTR
ARG3:   APTR    Buffer for new path string
ARG4:   LONG    Size of buffer in bytes

RES1:   LONG    Actual length of returned string, -2 if there isn't enough
space in buffer,or -1 for other errors
RES2:   CODE    Failure code
```

This action reads a link and returns a path name to the link's object. The link's name (plus any necessary path) is passed as a CPTR (ARG2) which points to a C-style string, *not a BSTR*. ACTION_READ_LINK returns the path name in ARG3. The length of the target string is returned in RES1 (or a -1 indicating an error).

2.0 only **ACTION_CHANGE_MODE 1028 ChangeMode(type,obj,mode)**

```
ARG1:   LONG    Type of object to change - either CHANGE_FH or CHANGE_LOCK
ARG2:   BPTR    object to be changed
ARG3:   LONG    New mode for object - see ACTION_FINDINPUT, and
ACTION_LOCATE_OBJECT

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 is DOSFALSE
```

This action requests that the handler change the mode of the given file handle or lock to the mode in ARG3. This request should fail if the handler can't change the mode as requested (for example an exclusive request for an object that has multiple users).

2.0 only **ACTION_COPY_DIR_FH 1030 DupLockFromFH(fh)**

```
ARG1:   BPTR    file handle

RES1:   BPTR    Lock associated with file handle or NULL
RES2:   CODE    Failure code if RES1 = NULL
```

This action requests that the handler return a lock associated with the currently opened file handle. The request may fail for any restriction imposed by the file system (for example when the file handle is not opened in a shared mode). The file handle is still usable after this call, unlike the lock in ACTION_FH_FROM_LOCK.

2.0 only **ACTION_PARENT_FH 1031 ParentOfFH(fh)**

```
ARG1:   BPTR    File handle to get parent of

RES1:   BPTR    Lock on parent of a file handle
RES2:   CODE    Failure code if RES1 = NULL
```

This action obtains a lock on the parent directory (or root of the volume if at the top level) for a currently opened file handle. The lock is returned as a shared lock and must be freed. Note that unlike ACTION_COPY_DIR_FH, the mode of the file handle is unimportant. For an open file, ACTION_PARENT_FH should return a lock under all circumstances.

2.0 only **ACTION_EXAMINE_ALL 1033 ExAll(lock,buff,size,type,ctl)**

```
ARG1:   BPTR    Lock on directory to examine
ARG2:   APTR    Buffer to store results
ARG3:   LONG    Length (in bytes) of buffer (ARG2)
ARG4:   LONG    Type of request - one of the following:
           ED_NAME Return only file names
           ED_TYPE Return above plus file type
           ED_SIZE Return above plus file size
           ED_PROTECTION Return above plus file protection
           ED_DATE Return above plus 3 longwords of date
           ED_COMMENT Return above plus comment or NULL
ARG5:   BPTR    Control structure to store state information.  The control
structure must be allocated with AllocDosObject()!

RES1:   LONG    Continuation flag - DOSFALSE indicates termination
RES2:   CODE    Failure code if RES1 is DOSFALSE
```

This action allows an application to obtain information on multiple directory entries. It is particularly useful for applications that need to obtain information on a large number of files and directories.

This action fills the buffer (ARG2) with partial or whole ExAllData structures. The size of the ExAllData structure depends on the type of request. If the request type field (ARG4) is set to ED_NAME, only the ed_Name field is filled in. Instead of copying the unused fields of the ExAllData structure into the buffer, ACTION_EXAMINE_ALL truncates the unused fields. This effect is cumulative, so requests to fill in other fields in the ExAllData structure causes all fields that appear in the structure *before* the requested field will be filled in as well. Like the ED_NAME case mentioned above, any field that appears after the requested field will be truncated (see the ExAllData structure below). For example, if the request field is set to ED_COMMENT, ACTION_EXAMINE_ALL fills in all the fields of the ExAllData structure, because the ed_Comment field is last. This is the only case where the packet returns entire ExAllData structures.

```
struct ExAllData {
        struct ExAllData *ed_Next;
        UBYTE  *ed_Name;
        LONG    ed_Type;
        ULONG   ed_Size;
        ULONG   ed_Prot;
        ULONG   ed_Days;
        ULONG   ed_Mins;
        ULONG   ed_Ticks;
        UBYTE  *ed_Comment;     /* strings will be after last used field */
};
```

Each ExAllData structure entry has an ead_Next field which points to the next ExAllData structure. Using these links, a program can easily chain through the ExAllData structures without having to worry about how large the structure is. *Do not examine the fields beyond those requested* as they certainly will not be initialized (and will probably overlay the next entry).

The most important part of this action is the ExAllControl structure. It ***must*** be allocated and freed through AllocDosObject()/FreeDosObject(). This allows the structure to grow if necessary with future revisions of the operating and file systems. Currently, ExAllControl contains four fields:

Entries - This field is maintained by the file system and indicates the actual number of entries present in the buffer after the action is complete. Note that a value of zero is possible here as no entries may match the match string.

LastKey - This field ***must*** be initialized to 0 by the calling application before using this packet for the first time. This field is maintained by the file system as a state indicator of the current place in the list of entries to be examined. The file system may test this field to determine if this is the first or a subsequent call to this action.

MatchString - This field points to a pattern matching string to control which directory entries are returned. If this field is NULL, then all entries are returned. Otherwise, this string is used to pattern match the names of all directory entries before putting them into the buffer. The default AmigaDOS pattern match routine is used unless MatchFunc is not NULL (see below). Note that it is not acceptable for the application to change this field between subsequent calls to this action for the same directory.

MatchFunc - This field contains a pointer to an alternate pattern matching routine to validate entries. If it is NULL then the standard AmigaDOS wild card routines will be used. Otherwise, MatchFunc points to a hook function that is called in the following manner:

```
BOOL = MatchFunc(hookptr, data,typeptr)
                  A0      A1    A2
hookptr    Pointer to hook being called
data       Pointer to (partially) filled in ExAll data for item being checked.
typeptr    Pointer to longword indicating the type of the ExAll request (ARG4).

This function is expected to return DOSTRUE if the entry is accepted and
DOSFALSE if it is to be discarded.
```

2.0 only

```
ACTION_EXAMINE_FH        1034    ExamineFH(fh,fib)
ARG1:   BPTR    File handle on open file
ARG2:   BPTR    FileInfoBlock to fill in

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 is DOSFALSE
```

This function examines a file handle and fills in the FileInfoBlock (found in ARG2) with information about the current state of the file. This routine is analogous to the ACTION_EXAMINE_OBJECT action for locks. Because it is not always possible to provide an accurate file size (for example when buffers have not been flushed or two processes are writing to a file), the fib_Size field (see *dos/dos.h*) may be inaccurate.

2.0 only

```
ACTION_ADD_NOTIFY        4097    StartNotify(NotifyRequest)
ARG1:   BPTR    NotifyRequest structure

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 is DOSFALSE
```

This action asks a file system to notify the calling program if a particular file is altered. A file system notifies a program either by sending a message or by signaling a task.

```
struct NotifyRequest {
        UBYTE *nr_Name;
        UBYTE *nr_FullName;             /* set by dos - don't touch */
        ULONG nr_UserData;              /* for applications use */
        ULONG nr_Flags;

        union {

            struct {
                struct MsgPort *nr_Port;        /* for SEND_MESSAGE */
            } nr_Msg;

            struct {
                struct Task *nr_Task;           /* for SEND_SIGNAL */
                UBYTE nr_SignalNum;             /* for SEND_SIGNAL */
                UBYTE nr_pad[3];
            } nr_Signal;
        } nr_stuff;

        ULONG nr_Reserved[4];           /* leave 0 for now */

        /* internal use by handlers */
        ULONG nr_MsgCount;              /* # of outstanding msgs */
        struct MsgPort *nr_Handler;     /* handler sent to (for EndNotify) */
};
```

To use this packet, an application needs to allocate and initialize a NotifyRequest structure (see above). As of this writing, NotifyRequest structures are not allocated by AllocDosObject(), but this may change in the future. The handler gets the watched file's name from the nr_FullName field. The current file system does not currently support wild cards in this field, although there is nothing to prevent other handlers from doing so.

The string in nr_FullName must be an absolute path, including the name of the root volume (no assigns). The absolute path is necessary because the file or its parent directories do not have to exist when the notification is set up. This allows notification on files in directories that do not yet exist. Notification will not occur until the directories and file are created.

An application that uses the StartNotify() DOS call does *not* fill in the NotifyRequest's nr_FullName field, but instead fills in the nr_Name field. StartNotify() takes the name from the nr_Name field and uses GetDeviceProc() and NameFromLock() to expand any assigns (such as ENV:), storing the result in nr_FullName. Any application utilizing the packet level interface instead of StartNotify() must expand their own assigns. Handlers must not count on nr_Name being correct.

The notification type depends on which bit is set in the NotifyRequest.nr_Flags field. If the `NRF_SEND_MESSAGE` bit is set, an application receives notification of changes to the file through a message (see NotifyMessage from *dos/notify.h*). In this case, the nr_Port field must point to the message port that will receive the notifying message . If the nr_Flags `NRF_SEND_SIGNAL` bit is set, the file system will signal a task instead of sending a message. In this case, nr_Task points to the task and nr_SignalNum is the signal number. *Only one of these two bits should be set!*

If a program sets the `NRF_WAIT_REPLY` bit, the handler must wait to send pending notifications until previous ones are returned. When a handler receives a notification request with the `NRF_NOTIFY_INITIAL` bit set, the handler sends an initial message or gives an initial signal if the watched file already exists.

Handlers should only perform a notification when the actual contents of the file have been changed. This includes `ACTION_WRITE`, `ACTION_TRUNCATE`, `ACTION_SET_DATE`, `ACTION_DELETE`, `ACTION_RENAME`, `ACTION_FINDUPDATE`, `ACTION_FINDINPUT`, and `ACTION_FINDOUTPUT`. It may also include other actions such as `ACTION_SET_COMMENT` or `ACTION_SET_PROTECT`, but this is not required (and may not be expected by the application as there is no need to reread the data).

2.0 only

ACTION_REMOVE_NOTIFY 4098 EndNotify(NotifyRequest)

```
ARG1:   BPTR    Pointer to previously added notify request

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 is DOSFALSE
```

This action cancels a notification (see `ACTION_ADD_NOTIFY`) . ARG1 is the NotifyRequest structure used to initiate the notification. The handler should abandon any pending notification messages. Note that it is possible for a file system to receive a reply from a previously sent notification message even after the notification has been terminated. It should accept these messages silently and throw them away.

Volume Manipulation/Information

The Volume Manipulation and Information actions are used to allow access to the underlying volume currently being manipulated by the file system.

```
ACTION_CURRENT_VOLUME     7       <sendpkt only>
RES1:   BPTR    Pointer to volume node of current volume
```

This action returns a pointer to the volume node (from the DOS device list) associated with the file system. As the volume node may be removed from the device list when the file system mounts a different volume (such as when directed to by an ACTION_INHIBIT) there is no guarantee that this pointer will remain valid for any amount of time. This action is generally used by AmigaDOS to provide the volume line of a requester.

```
ACTION_DISK_INFO          25      Info(...)
ARG1:   BPTR    Pointer to an InfoData structure to fill in

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)

ACTION_INFO               26      <sendpkt only>
ARG1:   LOCK    Lock
ARG2:   BPTR    Pointer to a InfoData Structure to fill in

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
```

These actions are used to get information about the device and status of the file handler. ACTION_DISK_INFO is used by the *info* command to report the status of the volume currently in the drive. It fills in an InfoData structure about the volume the file system currently controls. ACTION_INFO fills in an InfoData structure for the volume the lock (ARG1) is on instead of the volume currently in the drive. These actions are generally expected to return DOSTRUE.

The ACTION_DISK_INFO packet has a special meaning for console style handlers. When presented with this packet, a console style handler should return a pointer to the window associated with the open handle.

```
ACTION_RENAME_DISK        9       Relabel(...) in 2.0
ARG1:   BSTR    New disk name

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
```

This action allows an application to change the name of the current volume. A file system implementing this function must also change the name stored in the volume node of the DOS device list.

2.0 only

```
ACTION_FORMAT              1020    Format(fs,vol,type)
ARG1:    BSTR    Name of device (with trailing ':')
ARG2:    BSTR    Name for volume (if supported)
ARG3:    LONG    Type of format (file system specific)

RES1:    BOOL    Success/Failure (DOSTRUE/DOSFALSE)
RES2:    CODE    Failure code if RES1 is DOSFALSE
```

This packet tells a file system to perform any device or file system specific formatting on any newly initialized media. Upon receiving this action, a file system can assume that the media has already been low level formatted and should proceed to write out any high level disk structure necessary to create an empty volume.

Handler Maintenance and Control

A number of packets are defined to give an application some control over a file system:

```
ACTION_DIE                  5       <sendpkt only>
RES1:   BOOL    DOSTRUE
```

As its name implies, the ACTION_DIE packet tells a handler to quit. All new handlers are expected to implement this packet. Because of outstanding locks and the fact that the handler address is returned by the DeviceProc() routine, it is unlikely that the handler can disappear completely, but instead will have to release as many resources as possible and simply return an error on all packets sent to it.

In the future, the system may be able to determine if there are any outstanding DeviceProc() references to a handler, and therefore make it is safe to shut down completely.

```
ACTION_FLUSH                27      <sendpkt only>
RES1:   BOOL    DOSTRUE
```

This action causes the file system to flush out all buffers to disk *before* returning this packet. If any writes are pending, they must be processed before responding to this packet. This packet allows an application to make sure that the data that is supposed to be on the disk is actually written to the disk instead of waiting in a buffer.

```
ACTION_MORE_CACHE           18      AddBuffers(...) in 2.0
ARG1:   LONG    Number of buffers

RES1:   BOOL    DOSTRUE
RES2:   LONG    New number of buffers
```

This action allows an application to change the number of internal buffers used by the file system for caching. Note that a positive number increases the number of buffers while a negative number decreases the number of buffers. In all cases, the number of current buffers are returned in RES2. This allows an application to inquire the number of buffers by sending in a value of 0 (resulting in no change). Note that the OFS and FFS in 1.3 do not accept a negative number of buffers.

```
ACTION_INHIBIT              31      Inhibit(...) in 2.0
ARG1:   BOOL    DOSTRUE = inhibit,      DOSFALSE = uninhibit

RES1:   BOOL    Success/failure (DOSTRUE/DOSFALSE)
```

This action is probably one of the most dangerous that a file system has to handle. When inhibited (ARG1 = DOSTRUE), the file system must not access any underlying media and return an error code on all attempts to access the device. Once uninhibited (ARG1 = DOSFALSE), the file system must *assume* that the medium has been changed. The file system must flush the buffers before the ACTION_INHIBIT , popping up a requester demanding that the user put back the current disk, if necessary. The handler may choose to reject an inhibit request if any objects are open for writing.

Although it's not required, a handler should nest inhibits. Prior to 2.0, the system handlers did not keep a nesting count and were subject to some obscure race conditions. The 2.0 ROM filing system introduced a nesting count.

```
ACTION_WRITE_PROTECT     1023    <sendpkt only>
ARG1:   BOOL    DOSTRUE/DOSFALSE (write protect/un-write protect)
ARG2:   LONG    32 Bit pass key

RES1:   BOOL    DOSTRUE/DOSFALSE
```

This is a new packet defined for the Fast File System. This packet allows an application to change the write protect flag of a disk (if possible - applications cannot write to floppies that have their write-protect tabs set). This packet is primarily intended to allow write-protecting non-removable media such as hard disks. The value in ARG1 toggles the write status. The 32-bit passkey allows a program to prevent other programs from unwrite-protecting a disk. To unlock a disk, ARG2 must match the passkey of the packet that locked the disk, unless the disk was locked with a passkey of 0. In this case, no passkey is necessary to unlock the disk.

2.0 only

```
ACTION_IS_FILESYSTEM     1027    IsFileSystem(devname)

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 is DOSFALSE
```

Through this function, a handler can indicates whether or not it is a file system (whether or not it can support separate files for storing information). Programs will assume a handler can create multiple, distinct files through calls to Open() if the handler returns this packet with a `DOSTRUE` value. A handler does *not* need to support directories and subdirectories in order to qualify as a file system. It does have to support the Examine()/ExNext() calls.

Note that the AmigaDOS routine IsFileSystem() will attempt to use Lock(":",SHARED_ACCESS) if this packet returns `ERROR_ACTION_NOT_KNOWN`.

Handler Internal

There are several actions that are generally used by handlers to allow messages returning from requested services (typically an Exec device) to look like incoming request packets. This allows the handler to request an asynchronous operation but be notified of the completion. For example, a handler sends the *serial.device* a request for a read, but instead of sending a plain IO request, it sends a DOS packet disguised as an IO request. The *serial.device* treats the packet like a normal IO request, returning it to the handler when it is finished. When the handler gets back its disguised DOS packet, it knows that the read has completed.

ACTION_NIL **0** **<internal>**

Although not specifically an action, many returns look like this value because the action field has not been filled in.

ACTION_READ_RETURN **1001** **<internal>**

Generally used to indicate the completion of an asynchronous read request.

ACTION_WRITE_RETURN **1002** **<internal>**

Generally used to indicate the completion of an asynchronous write request.

ACTION_TIMER **30** **<internal>**

Used to indicate the passage of a time interval. Many handlers have a steady stream of ACTION_TIMER packets so that they can schedule house keeping and flush buffers when no activity has occurred for a given time interval.

Obsolete Packets

There are several packet types that are documented within the system include files that are obsolete. A file system is not expected to handle these packets and any program which sends these packets can not expect them to work:

```
ACTION_DISK_CHANGE      33      <Obsolete>

ACTION_DISK_TYPE        32      <Obsolete>

ACTION_EVENT             6      <Obsolete>

ACTION_GET_BLOCK         2      <Obsolete>

ACTION_SET_MAP           4      <Obsolete>
```

Of particular note here is `ACTION_DISK_CHANGE`. The *DiskChange* command uses the `ACTION_INHIBIT` packet to accomplish its task.

Console Only Packets

The remaining packets are only used for console handlers and do not need to be implemented by a file system.

```
ACTION_SCREEN_MODE      994     <sendpkt only>
ARG1:   LONG    Mode (zero or one)

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 is DOSFALSE
```

Switch the console to and from RAW mode. An ARG1 of one indicates the unprocessed, raw mode while an ARG1 of zero indicates the processed, "cooked" mode.

```
ACTION_WAIT_CHAR        20      WaitForChar()
ARG1:   ULONG   Timeout in microseconds

RES1:   BOOL    Success/Failure (DOSTRUE/DOSFALSE)
RES2:   CODE    Failure code if RES1 is DOSFALSE
```

Performs a timed read of a character. The WaitForChar() function uses this packet.

Summary of Defined Packet Numbers

This is a listing of all the DOS packets defined by Commodore. Packets 0-1999 are reserved for use by Commodore. Unless otherwise noted, packets 2050-2999 are reserved for use by third party developers (see chart below). The remaining packets are reserved for future expansion (Note: packets 2008, 2009, 4097, and 4098 are in use by Commodore).

```
Decimal Hex     Action #define

0       0x0000  ACTION_NIL
1               <Reserved by Commodore>
2       0x0002  ACTION_GET_BLOCK
3               <Reserved by Commodore>
4       0x0004  ACTION_SET_MAP
5       0x0005  ACTION_DIE
6       0x0006  ACTION_EVENT
7       0x0007  ACTION_CURRENT_VOLUME
8       0x0008  ACTION_LOCATE_OBJECT
9       0x0009  ACTION_RENAME_DISK
10-14           <Reserved by Commodore>
15      0x000F  ACTION_FREE_LOCK
16      0x0010  ACTION_DELETE_OBJECT
17      0x0011  ACTION_RENAME_OBJECT
18      0x0012  ACTION_MORE_CACHE
19      0x0013  ACTION_COPY_DIR
20      0x0014  ACTION_WAIT_CHAR
21      0x0015  ACTION_SET_PROTECT
22      0x0016  ACTION_CREATE_DIR
23      0x0017  ACTION_EXAMINE_OBJECT
24      0x0018  ACTION_EXAMINE_NEXT
25      0x0019  ACTION_DISK_INFO
26      0x001A  ACTION_INFO
27      0x001B  ACTION_FLUSH
28      0x001C  ACTION_SET_COMMENT
29      0x001D  ACTION_PARENT
30      0x001E  ACTION_TIMER
31      0x001F  ACTION_INHIBIT
32      0x0020  ACTION_DISK_TYPE
33      0x0021  ACTION_DISK_CHANGE
34      0x0022  ACTION_SET_DATE
35-39           <Reserved by Commodore>
40      0x0028  ACTION_SAME_LOCK
41-81           <Reserved by Commodore>
82      0x0052  ACTION_READ
83-86           <Reserved by Commodore>
87      0x0057  ACTION_WRITE
88-993          <Reserved by Commodore>
994     0x03E2  ACTION_SCREEN_MODE
995-1000        <Reserved by Commodore>
1001    0x03E9  ACTION_READ_RETURN
1002    0x03EA  ACTION_WRITE_RETURN
1003            <Reserved by Commodore>
1004    0x03EC  ACTION_FINDUPDATE
1005    0x03ED  ACTION_FINDINPUT
1006    0x03EE  ACTION_FINDOUTPUT
1007    0x03EF  ACTION_END
1008    0x03F0  ACTION_SEEK
1009-1019       <Reserved by Commodore>
```

1020	0x03FC	ACTION_FORMAT
1021	0x03FD	ACTION_MAKE_LINK
1022	0x03FE	ACTION_SET_FILE_SIZE
1023	0x03FF	ACTION_WRITE_PROTECT
1024	0x0400	ACTION_READ_LINK
1025		<Reserved by Commodore>
1026	0x0402	ACTION_FH_FROM_LOCK
1027	0x0403	ACTION_IS_FILESYSTEM
1028	0x0404	ACTION_CHANGE_MODE
1029		<Reserved by Commodore>
1030	0x0406	ACTION_COPY_DIR_FH
1031	0x0407	ACTION_PARENT_FH
1032		<Reserved by Commodore>
1033	0x0409	ACTION_EXAMINE_ALL
1034	0x040A	ACTION_EXAMINE_FH
1035-2007		<Reserved by Commodore>
2008	0x07D8	ACTION_LOCK_RECORD
2009	0x07D9	ACTION_FREE_RECORD
2010-2049		<Reserved by Commodore>
2050-2999		<Reserved for 3rd Party Handlers>
4097	0x1001	ACTION_ADD_NOTIFY
4098	0x1002	ACTION_REMOVE_NOTIFY
4099-		<Reserved by Commodore for Future Expansion>

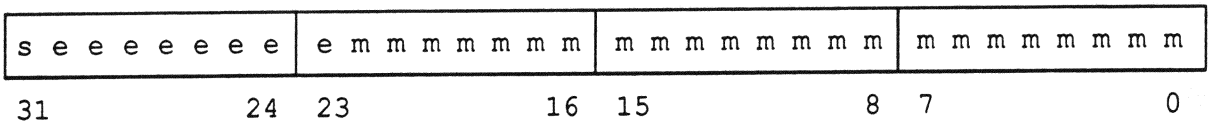
The DR2D FORM

by Ross Cunniff and John Orr

A standard IFF FORM to describe 2D drawings has been sorely needed for a long time. Several commercial drawing packages have been available for some time but none has established its file format as *the* Amiga standard. The absence of a 2D drawing standard hinders the development of applications that use 2D drawings as it forces each application to understand several private standards instead of a single one. Without a standard, data exchange for both the developer and the user is difficult, if not impossible.

The DR2D FORM fills this void. This FORM was developed by Taliesin, Inc. for use as the native file format for their two-dimensional structured drawing package, *ProVector*. Saxon Industries and Soft Logik Publishing Corporation are planning to support this new FORM in the near future.

Many of the values stored in the DR2D FORM are stored as IEEE single precision floating point numbers. These numbers consist of 32 bits, arranged as follows:



where:

- s is the sign of the number where 1 is negative and 0 is positive.
- e is the 8 bit exponent in excess 127 form. This number is the power of two to which the mantissa is raised (Excess 127 form means that 127 is added to the exponent before packing it into the IEEE number.)
- m is the 23 bit mantissa. It ranges from 1.0000000 to 1.999999..., where the leading base-ten one is assumed.

An IEEE single precision with the value of 0.0000000 has all its bits cleared.

The DR2D Chunks

FORM (0x464F524D) /* All drawings are a FORM */

```
struct FORMstruct {
    ULONG    ID;    /* DR2D */
    ULONG    Size;
};
```

DR2D (0x44523244) /* ID of 2D drawing */

The DR2D chunks are broken up into three groups: the global drawing attribute chunks, the object attribute chunks, and the object chunks. The global drawing attribute chunks describe elements of a 2D drawing that are common to many objects in the drawing. Document preferences, palette information, and custom fill patterns are typical document-wide settings defined in global drawing attribute chunks. The object attribute chunks are used to set certain properties of the object chunk(s) that follows the object attribute chunk. The current fill pattern, dash pattern, and line color are all set using an object attribute chunk. Object chunks describe the actual DR2D drawing. Polygons, text, and bitmaps are found in these chunks.

The Global Drawing Attribute Chunks

The following chunks describe global attributes of a DR2D document.

DRHD (0x44524844) /* Drawing header */

The DRHD chunk contains the upper left and lower right extremes of the document in (X, Y) coordinates. This chunk is required and should only appear once in a document in the outermost layer of the DR2D file (DR2Ds can be nested).

```
struct DRHDstruct {
    ULONG    ID;
    ULONG    Size;    /* Always 16 */
    IEEE XLeft, YTop,
         XRight, YBot;
};
```

The point (XLeft,YTop) is the upper left corner of the project and the point (XRight,YBot) is its lower right corner. These coordinates not only supply the size and position of the document in a coordinate system, they also supply the project's orientation. If XLeft < XRight, the X-axis increases toward the right. If YTop < YBot, the Y-axis increases toward the bottom. Other combinations are possible; for example in Cartesian coordinates, XLeft would be less than XRight but YTop would be *greater* than YBot.

PPRF (0x50505249) /* Page preferences */

The PPRF chunk contains preference settings for *ProVector*. Although this chunk is not required, its use is encouraged because it contains some important environment information.

```
struct PPRFstruct {
    ULONG    ID;
    ULONG    Size;
    char Prefs[Size];
};
```

DR2D stores preferences as a concatenation of several null-terminated strings, in the Prefs[] array. The strings can appear in any order. The currently supported strings are:

```
Units=<unit-type>
Portrait=<boolean>
PageType=<page-type>
GridSize=<number>
```

where:

`<unit-type>` is either `Inch`, `Cm`, or `Pica`
`<boolean>` is either `True` or `False`
`<page-type>` is either `Standard`, `Legal`, `B4`, `B5`, `A3`, `A4`, `A5`, or `Custom`
`<number>` is a floating-point number

The DR2D FORM does not require this chunk to explicitly state all the possible preferences. In the absence of any particular preference string, a DR2D reader should fall back on the default value. The defaults are:

```
Units=Inch
Portrait=True
PageType=Standard
GridSize=1.0
```

CMAP (0x434D4150) /* Color map (Same as ILBM CMAP) */

This chunk is identical to the ILBM CMAP chunk as described in the IFF ILBM documentation.

```
struct CMAPstruct {
    ULONG    ID;
    ULONG    Size;
    UBYTE    ColorMap[Size];
};
```

ColorMap is an array of 24-bit RGB color values. The 24-bit value is spread across three bytes, the first of which contains the red intensity, the next contains the green intensity, and the third contains the blue intensity. Because DR2D stores its colors with 24-bit accuracy, *DR2D readers must not make the mistake that some ILBM readers do in assuming the CMAP chunk colors correspond directly to Amiga color registers.*

FONS (0x464F4E53) /* Font chunk (Same as FTXT FONS chunk) */

The FONS chunk contains information about a font used in the DR2D FORM. *ProVector* does not include support for Amiga fonts. Instead, *ProVector* uses fonts defined in the OFNT FORM which is documented later in this article.

```
struct FONSstruct {
    ULONG     ID;
    ULONG     Size;
    UBYTE     FontID;         /* ID the font is referenced by */
    UBYTE     Pad1;           /* Always 0 */
    UBYTE     Proportional;   /* Is it proportional? */
    UBYTE     Serif;          /* does it have serifs? */
    CHAR      Name[Size-4];   /* The name of the font */
};
```

The UBYTE FontID field is the number DR2D assigns to this font. References to this font by other DR2D chunks are made using this number.

The Proportional and Serif fields indicate properties of this font. Specifically, Proportional indicates if this font is proportional, and Serif indicates if this font has serifs. These two options were created to allow for font substitution in case the specified font is not available. They are set according to these values:

0 The DR2D writer didn't know if this font is proportional/has serifs.
1 No, this font is not proportional/does not have serifs.
2 Yes, this font is proportional/does have serifs.

The last field, Name[], is a NULL terminated string containing the name of the font.

DASH (0x44415348) /* Line dash pattern for edges */

This chunk describes the on-off dash pattern associated with a line.

```
struct DASHstruct {
    ULONG     ID;
    ULONG     Size;
    USHORT    DashID;              /* ID of the dash pattern */
    USHORT    NumDashes;           /* Should always be even */
    IEEE      Dashes[NumDashes];   /* On-off pattern */
};
```

DashID is the number assigned to this specific dash pattern. References to this dash pattern by other DR2D chunks are made using this number.

The Dashes[] array contains the actual dash pattern. The first number in the array (element 0) is the length of the "on" portion of the pattern. The second number (element 1) specifies the "off" portion of the pattern. If there are more entries in the Dashes array, the pattern will continue. Even-index elements specify the length of an "on" span, while odd-index elements specify the length of an "off" span. There must be an even number of entries. *These lengths are not in the same units as specified in the PPRF chunk, but are multiples of the line width*, so a line of width 2.5 and a dash pattern of 1.0, 2.0 would have an "on" span of length 1.0 x 2.5 = 2.5 followed by an "off" span of length 2.0 x 2.5 = 5. The following figure shows several dash pattern examples. Notice that for lines longer than the dash pattern, the pattern repeats.

[3, 3] linewidth = 1

[3, 6] linewidth = 2.5

[1, 2, 3, 2] linewidth = 0.5

By convention, DashID 0 is reserved to mean 'No line pattern at all', i.e. the edges are invisible. This DASH pattern should *not* be defined by a DR2D DASH chunk. Again by convention, a NumDashes of 0 means that the line is solid.

AROW (0x41524F57) /* An arrow-head pattern */

The AROW chunk describes an arrowhead pattern. DR2D open polygons (OPLY) can have arrowheads attached to their endpoints. See the description of the OPLY chunk later in this article for more information on the OPLY chunk.

```
#define ARROW_FIRST  0x01  /* Draw an arrow on the OPLY's first point */
#define ARROW_LAST   0x02   /* Draw an arrow on the OPLY's last point */

struct AROWstruct {
    ULONG     ID;
    ULONG     Size;
    UBYTE     Flags;       /* Flags, from ARROW_*, above */
    UBYTE     Pad0;        /* Should always 0 */
    USHORT    ArrowID;     /* Name of the arrow head */
    USHORT    NumPoints;
    IEEE      ArrowPoints[NumPoints*2];
};
```

The Flags field specifies which end(s) of an OPLY to place an arrowhead based on the #defines above. ArrowID is the number by which an OPLY will reference this arrowhead pattern.

The coordinates in the array ArrowPoints[] define the arrowhead's shape. These points form a closed polygon. See the section on the OPLY/CPLY object chunks for a descriptionof how DR2D defines shapes. The arrowhead is drawn in the same coordinate system relative to the endpoint of the OPLY the arrowhead is attached to. The arrowhead's origin (0,0) coincides with the OPLY's endpoint. DR2D assumes that the arrowhead represented in the AROW chunk is pointing to the right so the proper rotation can be applied to the arrowhead. The arrow is filled according to the current fill pattern set in the ATTR object attribute chunk.

FILL (0x46494C4C) /* Object-oriented fill pattern */

The FILL chunk defines a fill pattern. This chunk is only valid inside nested DR2D FORMs. The GRUP object chunk section of this article contans an example of the FILL chunk.

```
struct FILLstruct {
    ULONG     ID;
    ULONG     Size;
    USHORT    FillID;  /* ID of the fill */
};
```

FillID is the number by which the ATTR object attribute chunk references fill patterns. The FILL chunk must be the first chunk inside a nested DR2D FORM. A FILL is followed by one DR2D object plus any of the object attribute chunks (ATTR, BBOX) associated with the object.

Fig. A

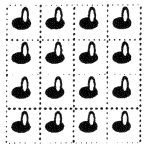


Fig. B

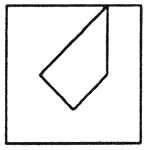


Fig. C

Fig. D

DR2Dmakes a "tile" out of the fill pattern, giving it a virtual bounding box based on the extreme X and Y values of the FILL's object (Fig. A). The bounding box shown in Fig. A surrounding the pattern (the two ellipses) is invisible to the user. In concept, this rectangle is pasted on the page left to right, top to bottom like floor tiles (Fig. B). Again, the bounding boxes are not visible. The only portion of this tiled pattern that is visible is the part that overlaps the object (Fig. C) being filled. The object's path is called a clipping path, as it "clips" its shape from the tiled pattern (Fig. D). Note that the fill is only *masked* on top of underlying objects, so any "holes" in the pattern will act as a window, leaving visible underlying objects.

LAYR (0x4C415952) /* Define a layer */

A DR2D project is broken up into one or more layers. Each DR2D object is in one of these layers. Layers provide several useful features. Any particular layer can be "turned off", so that the objects in the layer are not displayed. This eliminates the unnecessary display of objects not currently needed on the screen. Also, the user can lock a layer to protect the layer's objects from accidental changes.

```
struct LAYRstruct {
    ULONG     ID;
    ULONG     Size;
    USHORT    LayerID;          /* ID of the layer */
    char      LayerName[16];    /* Null terminated and padded */
    UBYTE     Flags;            /* Flags, from LF_*, below */
    UBYTE     Pad0;             /* Always 0 */
};
```

LayerID is the number assigned to this layer. As the field's name indicates, LayerName[] is the NULL terminated name of the layer. Flags is a bit field who's bits are set according to the #defines below:

```
#define LF_ACTIVE     0x01  /* Active for editing */
#define LF_DISPLAYED  0x02  /* Displayed on the screen */
```

If the LF_ACTIVE bit is set, this layer is unlocked. A set LF_DISPLAYED bit indicates that this layer is currently visible on the screen. A cleared LF_DISPLAYED bit implies that LF_ACTIVE is not set. The reason for this is to keep the user from accidentally editing layers that are invisible.

The Object Attribute Chunks

ATTR (0x41545452) /* Object attributes */

The ATTR chunk sets various attributes for the objects that follow it. The attributes stay in effect until the next ATTR changes the attributes, or the enclosing FORM ends, whichever comes first.

```
/* Various fill types */
#define FT_NONE     0    /* No fill*/
#define FT_COLOR    1    /* Fill with color from palette */
#define FT_OBJECTS  2    /* Fill with tiled objects*/


struct ATTRstruct {
    ULONG     ID;
    ULONG     Size;
    UBYTE     FillType;      /* One of FT_*, above*/
    UBYTE     JoinType;      /* One of JT_*, below*/
    UBYTE     DashPattern;   /* ID of edge dash pattern*/
    UBYTE     ArrowHead;     /* ID of arrowhead to use */
    USHORT    FillValue;     /* Color or object with which to fill */
    USHORT    EdgeValue;     /* Edge color index*/
    USHORT    WhichLayer;    /* ID of layer it's in*/
    IEEE      EdgeThick;     /* Line width*/
};
```

FillType specifies what kind of fill to use on this ATTR chunk's objects. A value of FT_NONE means that this ATTR chunk's objects are not filled. FT_COLOR indicates that the objects should be filled in with a color. That color's ID (from the CMAP chunk) is stored in the FillValue field. If FillType is equal to FT_OBJECTS, FillValue contains the ID of a fill pattern defined in a FILL chunk.

JoinType determines which style of line join to use when connecting the edges of line segments. The field contains one of these four values:

```
/* Join types */
#define JT_NONE   0    /* Don't do line joins */
#define JT_MITER  1    /* Mitered join */
#define JT_BEVEL  2    /* Beveled join */
#define JT_ROUND  3    /* Round join */
```

DashPattern and ArrowHead contain the ID of the dash pattern and arrow head for this ATTR's objects. A DashPattern of zero means that there is no dash pattern so lines will be invisible. If ArrowHead is 0, OPLYs have no arrow head. EdgeValue is the color of the line segments. WhichLayer contains the ID of the layer this ATTR's objects are in. EdgeThick is the width of this ATTR's line segments.

BBOX (0x42424F48) /* Bounding box of next object in FORM */

The BBOX chunk supplies the dimensions and position of a bounding box surrounding the DR2D object that follows this chunk in the FORM. A BBOX chunk can apply to a FILL or AROW as well as a DR2D object. The BBOX chunk appears just before its DR2D object, FILL, or AROW chunk.

```
struct BBOXstruct {
    ULONG     ID;
    ULONG     Size;
    IEEE      XMin, YMin,  /* Bounding box of obj. */
              XMax, YMax;  /* including line width*/
};
```

In a Cartesian coordinate system, the point (XMin, YMin) is the coordinate of the lower left hand corner of the bounding box and (XMax, YMax) is the upper right. These coordinates take into consideration the width of the lines making up the bounding box.

XTRN (0x5854524E) /* Externally controlled object */

The XTRN chunk was created primarily to allow *ProVector* to link DR2D objects to ARexx functions.

```
struct XTRNstruct {
    ULONG     ID;
    ULONG     Size;
    short     ApplCallBacks;              /* From #defines, below */
    short     ApplNameLength;
    char      ApplName[ApplNameLength];   /* Name of ARexx func to call */
};
```

ApplName[] contains the name of the ARexx script *ProVector* calls when the user manipulates the object in some way. The ApplCallBacks field specifies the particular action that triggers calling the ARexx script according to the #defines listed below.

```
/* Flags for ARexx script callbacks */
#define    X_CLONE     0x0001    /* The object has been cloned */
#define    X_MOVE      0x0002    /* The object has been moved */
#define    X_ROTATE    0x0004    /* The object has been rotated */
#define    X_RESIZE    0x0008    /* The object has been resized */
#define    X_CHANGE    0x0010    /* An attribute (see ATTR) of the
                                    object has changed */
#define    X_DELETE    0x0020    /* The object has been deleted */
#define    X_CUT       0x0040    /* The object has been deleted, but
                                    stored in the clipboard */
#define    X_COPY      0x0080    /* The object has been copied to the
                                    clipboard */
#define    X_UNGROUP   0x0100    /* The object has been ungrouped */
```

For example, given the XTRN object:

```
FORM xxxx DR2D {
     XTRN xxxx { X_RESIZE | X_MOVE, 10, "Dimension" }
     ATTR xxxx { 0, 0, 1, 0, 0, 0, 0.0 }
     FORM xxxx DR2D {
          GRUP xxxx { 2 }
          STXT xxxx { 0, 0.5, 1.0, 6.0, 5.0, 0.0, 4, "3.0" }
          OPLY xxxx { 2, { 5.5, 5.5, 8.5, 5.5 } }
     }
}
```

ProVector would call the ARexx script named *Dimension* if the user resized or moved this object. What exactly *ProVector* sends depends upon what the user does to the object. The following list shows what string(s) *ProVector* sends according to which flag(s) are set. The parameters are described below.

```
X_CLONE     "appl CLONE objID dx dy"
X_MOVE      "appl MOVE objID dx dy"
X_ROTATE    "appl ROTATE objID cx cy angle"
X_RESIZE    "appl RESIZE objID cx cy sx sy"
X_CHANGE    "appl CHANGE objID et ev ft fv ew jt fn"
X_DELETE    "appl DELETE objID"
X_CUT       "appl CUT objID"
X_COPY      "appl COPY objID"
X_UNGROUP   "appl UNGROUP objID"
```

where:

`appl` is the name of the ARexx script
`CLONE`, `MOVE`, `ROTATE`, `RESIZE`, etc. are literal strings
`objID` is the object ID that *ProVector* assigns to this object
(`dx`, `dy`) is the position offset of the CLONE or MOVE
(`cx`, `cy`) is the point around which the object is rotated or resized
`angle` is the angle (in degrees) the object is rotated
`sx` and `sy` are the scaling factors in the horizontal and vertical directions, respectively.
`et` is the edge type (the dash pattern index)
`ev` is the edge value (the edge color index)
`ft` is the fill type
`fv` is the fill index
`ew` is the edge weight
`jt` is the join type
`fn` is the font name

The `X_CHANGE` message reflects changes to the attributes found in the ATTR chunk.

If the user resized the XTRN object shown above by factor of 2, *ProVector* would call the ARexx script *Dimension* like this:

```
Dimension RESIZE 1985427 7.0 4.75 2.0 2.0
```

The Object Chunks

The following chunks define the objects available in the DR2D FORM.

VBM (0x56424D20) /* Virtual BitMap */

The VBM chunk contains the position, dimensions, and file name of an ILBM image.

```
struct VBMstruct {
    IEEE      XPos, YPos,      /* Virtual coords */
              XSize, YSize,    /* Virtual size */
              Rotation;        /* in degrees */
    USHORT    PathLen;         /* Length of dir path */
    char      Path[PathLen];   /* Null-terminated path of file */
};
```

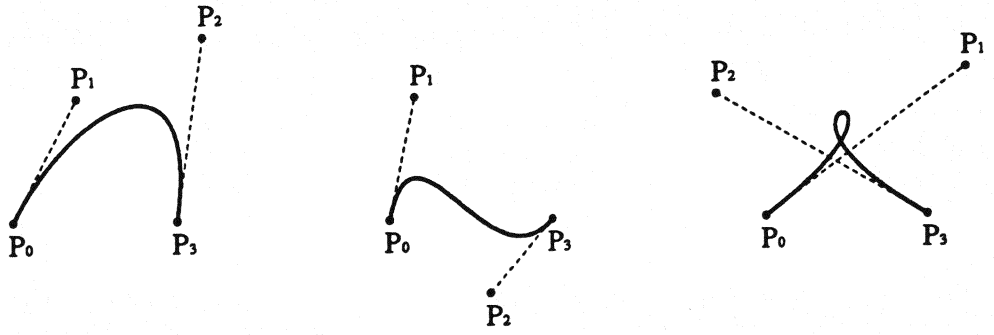
The coordinate (XPos, YPos) is the position of the upper left hand corner of the bitmap and the XSize and YSize fields supply the x and y dimensions to which the image should be scaled. Rotation tells how many degrees to rotate the ILBM around its upper left hand corner. *ProVector* does not currently support rotation of bitmaps and will ignore this value. Path contains the name of the ILBM file and may also contain a partial or full path to the file. DR2D readers should not assume the path is correct. The full path to an ILBM on one system may not match the path to the same ILBM on another system. If a DR2D reader cannot locate an ILBM file based on the full path name or the file name itself (looking in the current directory), it should ask the user where to find the image.

CPLY (0x43504C59) /* Closed polygon */
OPLY (0x4F504C59) /* Open polygon */

Polygons are the basic components of almost all 2D objects in the DR2D FORM. Lines, squares, circles, and arcs are all examples of DR2D polygons. There are two types of DR2D polygons, the open polygon (OPLY) and the closed polygon (CPLY). The difference between a closed and open polygon is that the computer adds a line segment connecting the endpoints of a closed polygon so that it is a continuous path. An open polygon's endpoints do not have to meet, like the endpoints of a line segment.

```
struct POLYstruct {
    ULONG     ID;
    ULONG     Size;
    USHORT    NumPoints;
    IEEE PolyPoints[2*NumPoints];
};
```

The NumPoints field contains the number of points in the polygon and the PolyPoints array contains the (X, Y) coordinates of the points of the non-curved parts of polygons. The even index elements are X coordinates and the odd index elements are Y coordinates.



DR2D uses Bézier cubic sections, or cubic splines, to describe curves in polygons. A set of four coordinates (P_1 through P_4) defines the shape of a cubic spline. The first coordinate (P_1) is the point where the curve begins. The line from the first to the second coordinate (P_1 to P_2) is tangent to the curve at the first point. The line from P_3 to P_4 is tangent to the cubic section, where it ends at P_4.

The coordinates describing the cubic section are stored in the PolyPoints[] array with the coordinates of the normal points. DR2D inserts an indicator point before a set of cubic section points to differentiate a normal point from the points that describe a curve. An indicator point has an X value of 0xFFFFFFFF. The indicator point's Y value is a bit field. If this bit field's low-order bit is set, the points that follow the indicator point make up a cubic section.

The second lowest order bit in the indicator point's bit field is the MOVETO flag. If this bit is set, the point (or set of cubic section points) starts a new polygon, or subpolygon. This subpolygon will appear to be completely separate from other polygons but there is an important connection between a polygon and its subpolygon. Subpolygons make it possible to create holes in polygons. An example of a polygon with a hole is the letter "O". The "O" is a filled circular polygon with a smaller circular polygon within it. The reason the inner polygon isn't covered up when the outer polygon is filled is that DR2D fills are done using the *even-odd rule.*

The even-odd rule determines if a point is "inside" a polygon by drawing a ray outward from that point and counting the number of path segments the ray crosses. If the number is even, the point is outside the object and shouldn't be filled. Conversely, an odd number of crossings means the point is inside and should be filled. DR2D only applies the even-odd rule to a polygon and its subpolygons, so no other objects are considered in the calculations.

Taliesin, Inc. supplied the following algorithm to illustrate the format of DR2D polygons. OPLYs, CPLYs, AROWs, and *ProVector*'s outline fonts all use the same format:

```
typedef union {
    IEEE num;
    LONG bits;
} Coord;

#define INDICATOR       0xFFFFFFFF
#define IND_SPLINE      0x00000001
#define IND_MOVETO      0x00000002

/* A common pitfall in attempts to support DR2D has
       been to fail to recognize the case when an
       INDICATOR point indicates the following
```

```
        coordinate to be the first point of BOTH a
        Bezier cubic and a sub-polygon, ie. the
        value of the flag = (IND_CURVE | IND_MOVETO) */

Coord   Temp0, Temp1;
int     FirstPoint, i, Increment;

/* Initialize the path */
NewPath();
FirstPoint = 1;

/* Draw the path */
i = 0;
while( i < NumPoints ) {
    Temp0.num = PolyPoints[2*i];    Temp1.num = PolyPoints[2*i + 1];
    if( Temp0.bits == INDICATOR ) {
        /* Increment past the indicator */
        Increment = 1;
        if( Temp1.bits & IND_MOVETO ) {
            /* Close and fill, if appropriate */
            if( ID == CPLY ) {
                FillPath();
            }
            else {
                StrokePath();
            }

            /* Set up the new path */
            NewPath();
            FirstPoint = 1;
        }
        if( Temp1.bits & IND_CURVE ) {
            /* The next 4 points are Bezier cubic control points */
            if( FirstPoint )
                MoveTo( PolyPoints[2*i + 2], PolyPoints[2*i + 3] );
            else
                LineTo( PolyPoints[2*i + 2], PolyPoints[2*i + 3] );
            CurveTo(    PolyPoints[2*i + 4], PolyPoints[2*i + 5],
                        PolyPoints[2*i + 6], PolyPoints[2*i + 7],
                        PolyPoints[2*i + 8], PolyPoints[2*i + 9] );
            FirstPoint = 0;
            /* Increment past the control points */
            Increment += 4;
        }
    }
    else {
        if( FirstPoint )
            MoveTo(     PolyPoints[2*i], PolyPoints[2*i + 1] );
        else
            LineTo(     PolyPoints[2*i], PolyPoints[2*i + 1] );
        FirstPoint = 0;

        /* Increment past the last endpoint */
        Increment = 1;
    }

    /* Add the increment */
    i += Increment;
}

/* Close the last path */
if( ID == CPLY ) {
    FillPath();
}
else {
    StrokePath();
}
```

GRUP (0x47525550) /* Group */

The GRUP chunk combines several DR2D objects into one. This chunk is only valid inside nested DR2D FORMs, and must be the first chunk in the FORM.

```
struct GROUPstruct {
    ULONG     ID;
    ULONG     Size;
    USHORT    NumObjs;
};
```

The NumObjs field contains the number of objects contained in this group. Note that the layer of the GRUP FORM overrides the layer of objects within the GRUP. The following example illustrates the layout of the GRUP (and FILL) chunk.

```
  FORM { DR2D              /* Top-level drawing... */
         DRHD { ... }       /* Confirmed by presence of DRHD chunk */
         CMAP { ... }       /* Various other things... */
         FONS { ... }
         FORM { DR2D              /* A nested form... */
                FILL { 1 }        /* Ah!  The fill-pattern table */
                CPLY { ... }      /* with only 1 object */
         }
         FORM { DR2D              /* Yet another nested form */
                GRUP { ..., 3 } /* Ah! A group of 3 objects */
                TEXT { ... }
                CPLY { ... }
                OPLY { ... }
         }
         FORM { DR2D              /* Still another nested form */
                GRUP { ..., 2 } /* A GRUP with 2 objects */
                OPLY { ... }
                TEXT { ... }
         }
  }
```

STXT (0x53545854) /* Simple text */

The STXT chunk contains a text string along with some information on how and where to render the text.

```
struct STXTstruct {
    ULONG     ID;
    ULONG     Size;
    UBYTE     Pad0;           /* Always 0 (for future expansion) */
    UBYTE     WhichFont;      /* Which font to use */
    IEEE      CharW, CharH,   /* W/H of an individual char */
              BaseX, BaseY,   /* Start of baseline */
              Rotation;       /* Angle of text (in degrees) */
    USHORT    NumChars;
    char      TextChars[NumChars];
};
```

The text string is in the character array, TextChars[]. The ID of the font used to render the text is WhichFont. The font's ID is set in a FONS chunk. The starting point of the baseline of the text is (BaseX, BaseY). This is the point around which the text is rotated. If the Rotation field is zero (degrees), the text's baseline will originate at (BaseX, BaseY) and move to the right. CharW and CharH are used to scale the text after rotation. CharW is the average character width and CharH is the average character height. The CharW/H fields are comparable to an X and Y font size.

TPTH (0x54505448) /* A text string along a path */

This chunk defines a path (polygon) and supplies a string to render along the edge of the path.

```
struct TPTHstruct {
    ULONG     ID;
    ULONG     Size;
    UBYTE     Justification;        /* see defines, below */
    UBYTE     WhichFont;            /* Which font to use */
    IEEE      CharW, CharH;         /* W/H of an individual char*/
    USHORT    NumChars;             /* Number of chars in the string */
    USHORT    NumPoints;            /* Number of points in the path */
    char      TextChars[NumChars];  /* PAD TO EVEN #! */
    IEEE      Path[2*NumPoints];    /* The path on which the text lies */
};
```

WhichFont contains the ID of the font used to render the text. Justification controls how the text is justified on the line. Justification can be one of the following values:

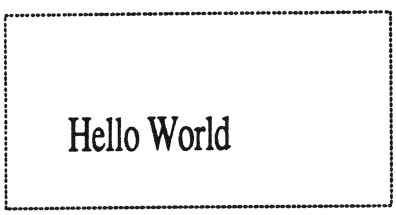
```
#define J_LEFT    0x00    /* Left justified */
#define J_RIGHT   0x01    /* Right justified */
#define J_CENTER  0x02    /* Center text */
#define J_SPREAD  0x03    /* Spread text across path */
```

CharW and CharH are the average width and height of the font characters and are akin to X and Y font sizes, respectively. A negative FontH implies that the font is upsidedown. Note that CharW must not be negative. NumChars is the number of characters in the TextChars[] string, the string containing the text to be rendered. NumPoints is the number of points in the Path[] array. Path[] is the path along which the text is rendered. The path itself is not rendered. The points of Path[] are in the same format as the points of a DR2D polygon.

A Simple DR2D Example

Here is a (symbolic) DR2D FORM:

```
FORM { DR2D
        DRHD 16 { 0.0, 0.0, 10.0, 8.0 }
        CMAP  6 { 0,0,0, 255,255,255 }
        FONS  9 { 1, 0, 1, 0, "Roman" } 0
        DASH 12 { 1, 2, {1.0, 1.0} }
        ATTR 14 { 0, 0, 1, 0, 0, 0, 0, 0.0 }
        BBOX 16 { 2.0, 2.0, 8.0, 6.0 }
        FORM { DR2D
                GRUP  2 { 2 }
                BBOX 16 { 3.0, 4.0, 7.0, 5.0 }
                STXT 36 { 0,1, 0.5, 1.0, 3.0, 5.0, 0.0, 12, "Hello, World" }
                BBOX 16 { 2.0, 2.0, 8.0, 6.0 }
                OPLY 42 { 5, {2.0,2.0, 8.0,2.0, 8.0,6.0, 2.0,6.0, 2.0,2.0 }
        }
}
```



This is what the DR2D FORM above should look like.

The OFNT FORM

OFNT (0x4F464E54) /* ID of outline font file */

ProVector's outline fonts are stored in an IFF FORM called OFNT. This IFF is a separate file from a DR2D. DR2D's FONS chunk refers only to fonts defined in the OFNT form.

OFHD (0x4F464844) /* ID of OutlineFontHeaDer */

This chunk contains some basic information on the font.

```
struct OFHDstruct {
    char      FontName[32];    /* Font name, null padded */
    short     FontAttrs;       /* See FA_*, below */
    IEEE      FontTop,         /* Typical height above baseline */
              FontBot,         /* Typical descent below baseline */
              FontWidth;       /* Typical width, i.e. of the letter O */
};

#define FA_BOLD       0x0001
#define FA_OBLIQUE    0x0002
#define FA_SERIF      0x0004
```

The FontName field is a NULL terminated string containing the name of this font. FontAttrs is a bit field with flags for several font attributes. The flags, as defined above, are bold, oblique, and serif. The unused higher order bits are reserved for later use. The other fields describe the average dimensions of the characters in this font. FontTop is the average height above the baseline, FontBot is the average descent below the baseline, and FontWidth is the average character width.

KERN (0x4B45524C) /* Kerning pair */

The KERN chunk describes a kerning pair. A kerning pair sets the distance between a specific pair of characters.

```
struct KERNstruct {
    short     Ch1, Ch2;    /* The pair to kern (allows for 16 bits...) */
    IEEE      XDisplace,   /* Amount to displace -left +right */
              YDisplace;   /* Amount to displace -down +up */
};
```

The Ch1 and Ch2 fields contain the pair of characters to kern. These characters are typically stored as ASCII codes. Notice that OFNT stores the characters as a 16-bit value. Normally, characters are stored as 8-bit values. The wary programmer will be sure to cast assigns properly to avoid problems with assigning an 8-bit value to a 16-bit variable. The remaining fields, XDisplace and YDisplace, supply the baseline shift from Ch1 to Ch2.

CHDF (0x43484446) /* Character definition */

This chunk defines the shape of *ProVector*'s outline fonts.

```
struct CHDFstruct {
    short     Ch;          /* The character we're defining (ASCII) */
    short     NumPoints;   /* The number of points in the definition */
    IEEE      XWidth,      /* Position for next char on baseline - X */
              YWidth;      /* Position for next char on baseline - Y */
    /* IEEE   Points[2*NumPoints]*/    /* The actual points */
};


#define  INDICATOR    0xFFFFFFFF   /* If X == INDICATOR, Y is an action */
#define  IND_SPLINE   0x00000001   /* Next 4 pts are spline control pts */
#define  IND_MOVETO   0x00000002   /* Start new subpoly */
#define  IND_STROKE   0x00000004   /* Stroke previous path */
#define  IND_FILL     0x00000008   /* Fill previous path */
```

Ch is the value (normally ASCII) of the character outline this chunk defines. Like Ch1 and Ch2 in the KERN chunk, Ch is stored as a 16-bit value. (XWidth,YWidth) is the offset to the baseline for the following character. OFNT outlines are defined using the same method used to define DR2D's polygons (see the description of OPLY/CPLY for details).

Because the OFNT FORM does not have an ATTR chunk, it needed an alternative to make fills and strokes possible. There are two extra bits used in font indicator points not found in polygon indicator points, the `IND_STROKE` and `IND_FILL` bits (see defines above). These two defines describe how to render the current path when rendering fonts.

The current path remains invisible until the path is either filled and/or stroked. When the `IND_FILL` bit is set, the currently defined path is filled in with the current fill pattern (as specified in the current ATTR chunk). A set `IND_STROKE` bit indicates that the currently defined path itself should be rendered. The current ATTR's chunk dictates the width of the line, as well as several other attributes of the line. These two bits apply only to the OFNT FORM and should not be used in describing DR2D polygons.

Amiga Mail
Volume II
November/December 1990

From the Desk of Jeff Scherb

At the end of 1990, Commodore made a few changes to its international organization which will improve our ability to assist developers in entering foreign markets. One of these changes include the creation of CATS-Europe. CATS-Europe will help to unify Commodore's international developer support efforts.

Commodore's developer support is generally handled on a local level by its sales subsidiaries. Commodore has separate local sales subsidiaries each of which communicates with Commodore International Limited (CIL). Subsidiaries exist for most of the major countries in Europe, as well as the U.S. and Canada. The CATS tech support folks in West Chester normally support only U.S. developers. Other sales companies have their own developer support programs, but they use the CATS developer publications and support materials, like Amiga Mail. In turn, each country contributes to Amiga Mail and Amiga Mail Market, and provides CATS with updated lists of local developers, so CATS can maintain an up-to-date world-wide developer list.

We'll be entering 1991 with a new structure -- Commodore people formerly involved with the European developer program (ESCO) have become CATS-Europe. CATS-Europe is based in Frankfurt, Germany, and reports to CATS world headquarters in the United States. The main mission of CATS-Europe is to better support European companies' marketing efforts in both the U.S. and Europe as well as U.S. companies' marketing efforts in Europe. Europe is a much larger market for Commodore than the U.S., but it is often difficult for small U.S. developers to get quality distribution in Europe. We're planning a number of events to put European distributors in touch with U.S. developers. We'll also increase European market coverage to the North American developer conference.

Leading CATS-Europe is Wolfgang Trompetter. Wolfgang comes to us from Ashton-Tate, where he held a number of positions, including Manager of Sales Support, Sales Manager for Dealer Sales, and most recently, Manager, Applications Product Group.

Gunda O'Neal, who many of you already know, will continue in her role of providing coordination and liaison between the CATS developer support program in the United States and the country support programs in Europe. Gunda will also continue to be the primary planner of the European Amiga Developer Conferences.

Wolfgang and I will be holding a special session at the Milan DevCon to gather feedback from developers regarding the ways we can assist you with the European market. I hope to see you in Milan, but if you can't be there, and have ideas for CATS-Europe, please pass them along to Wolfgang or me. We're anxious to do everything we can to help you enlarge your markets.

-JS